

Creating an ECO business model

Table of Contents

Overview	1
Prerequisites and goals	2
Terminology in ECO and UML	3
Creating an ECO model	5
Understanding the project template	7
Creating a class diagram	10
Adding attributes to the class	11
Adding associations between classes	13
Adding object validation	16
Generating source code	19
Inspecting the source code	19
Derived members	23
Creating OCL derived members	23
Creating code derived members	24
Creating derived settable members	26
Default string representation	29
Summary	30
Index	a

1 Overview

Although it is possible to add your business classes to your UI application it is recommended that you instead create a class library. This will reduce the possibility of cross contaminating source code in your business and presentation layers, which might prevent the same business model from being used by a secondary user interface such as ASP .NET.

2 Prerequisites and goals

Prerequisites

To successfully follow this document the user should have ECO installed. An understanding of UML would be an advantage.

Goals

By the end of this document you will be able to


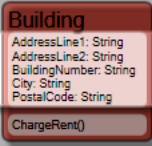
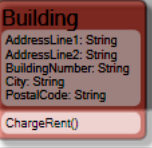
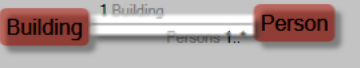
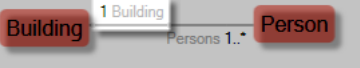
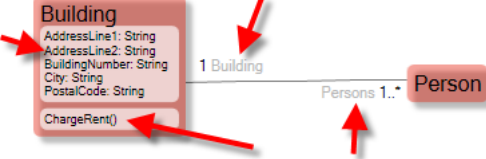
- Create a basic ECO model.
- Added ECO classes.
- Add UML attributes (aka properties) to those classes.
- Add associations between classes to allow them to hold references to each other.
- Add object validation expressions using constraints.
- Inspect generated source code and understand the additional .NET attributes generated.

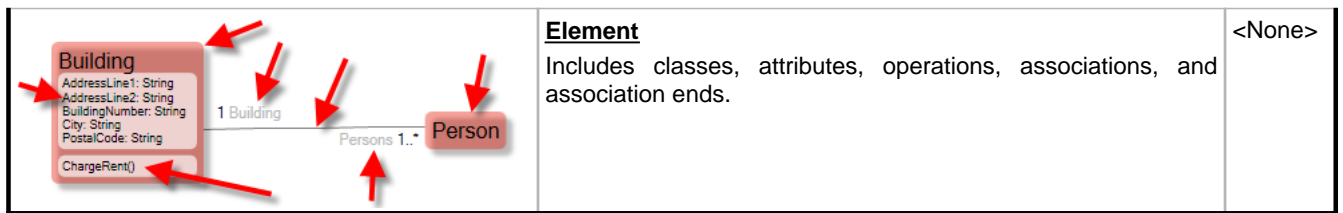
3 Terminology in ECO and UML

This section describes the different terminologies used between UML, source code, and EcoModeler. It is useful to understand when reading these documents but might initially cause some confusion. The following information might prove useful for later reference if it isn't clear after your first read through.

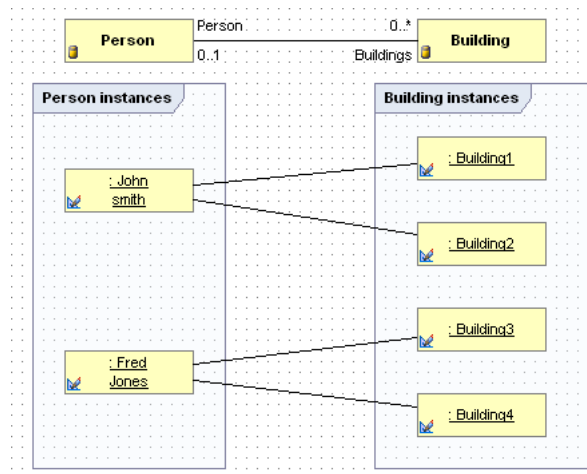
Object oriented terminology can at time be confusing. You may at times encounter different words used to describe the same thing depending on which software application you use, which part of the development stage you are at (design or implementation), and so on.

On the whole ECO uses the UML standard as it is a framework based on UML principles, as a consequence these articles will use terms such as "attribute", "operation", "association", "association end", "member", and "element". At various points however these articles may use the **Source code name** as illustrated in the following table.

Illustration	Description	Source code name
	Class Identifies a business object type. Instances of this type may be created at runtime and persisted to the data storage.	Class
	Attribute This identifies an item within a class definition such as FirstName, LastName, or DateOfBirth. These are usually simple types such as strings or integers, although ECO does allow you to use custom types.	Property
	Operation A procedural routine which an object may be instructed to perform.	Method
	Association A way of identifying a relationship between two classes.	<None>
	Association end An object's way of navigating to the instance(s) at the opposite end of the association.	Property
	Member An operation, attribute, or association end within a class.	Property / Method

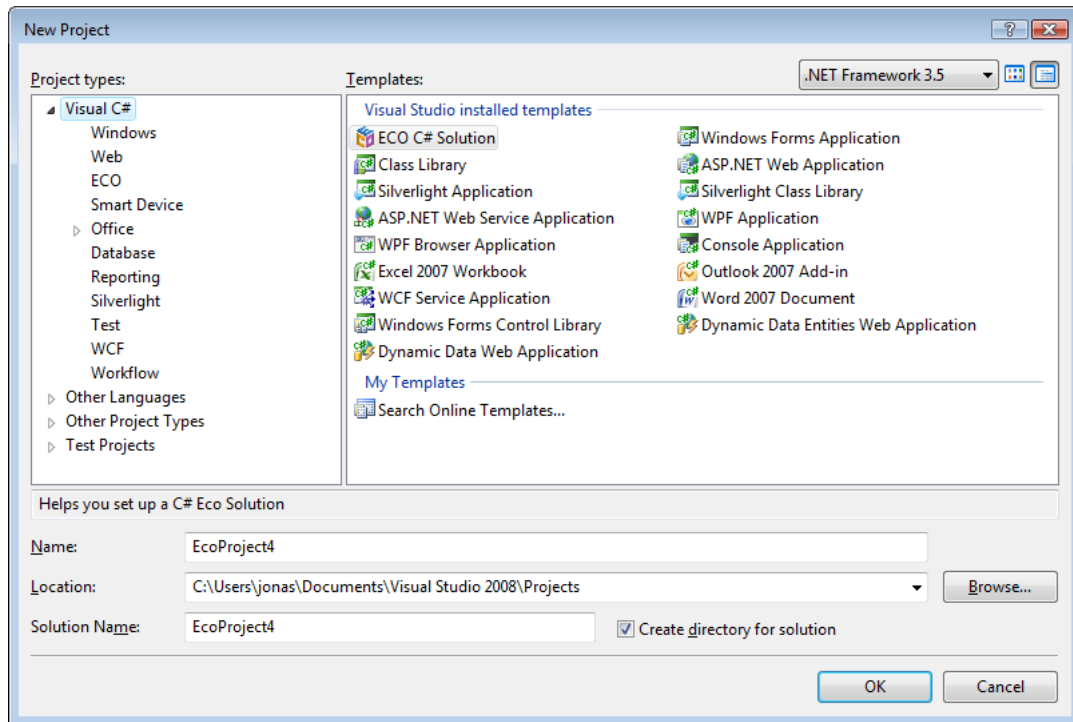


The preceding elements all define your model. Once your model is defined and your source code has been generated it is possible to create instances of your modeled classes at runtime.



4 Creating an ECO model

1. Start Microsoft Visual Studio.
1. File->New->Project.
2. Select the Visual C# node in the TreeView and select "ECO C# Solution" in templates window to the right.
3. Name the project "QuickStart" and click "OK".



4. Select "WebForm Application" from the drop down list and check the "Windows Forms" check box.
5. Under Model, enter QuickStartPackage for the model name.
6. Set the Persistence Storage option to "Xml".
7. Click "OK".



You now have a project containing an ECO business model that when compiled will create a DLL which may be used in multiple projects, next some business classes will be added to the model.

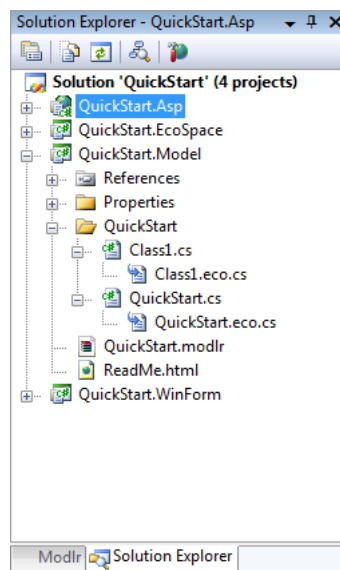
5 Understanding the project template

After completing the wizard steps you will be presented with a solution containing four projects. In alphabetical order these are:

- QuickStart.ASP
- QuickStart.EcoSpace
- QuickStart.Model
- QuickStart.WinForms

QuickStart.Model

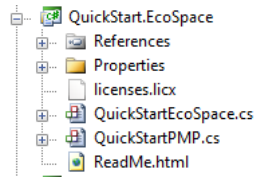
This project will contain the source code for your business classes; such as Person, Company, CustomerRole, and so on. The classes will all belong to what is known as a UML Package, a logical collection of classes. Although these classes belong to a single UML Package it is possible to use more than one UML Package in your application; it is also possible to reference other UML Packages and reuse the same UML Package across multiple applications as you will see during this series.



Filename	Description
Class1.cs	A default class added to the model. When the model is edited the first action you are most likely to take will be to rename this class in the modeler. The name of the class will be renamed in the project when you update the source code from the modeler.
Class1.eco.cs	This file contains the framework specific parts of the partial class Class1

QuickStart.modlr	This file is the design time UML representation of the UML Package. Modifications may be made to this file using the model designer. Changes to the source code (implementation) of the UML Package are not made unless the developer performs a code-generation operation within the modeler. This allows the developer to work with the model until they are happy with their results without affecting the project's source code.
QuickStart.cs	This is a meta-class. Its purpose is to describe information about the package, including which classes it contains. This class is used at runtime by ECO when it needs to determine model information.
QuickStart.eco.cs	This file contains the framework specific implementation for the package class

QuickStart.EcoSpace



An EcoSpace is a kind of service layer for your business classes, it is a container for both meta-information about the model and also instances of the modeled business classes. At runtime ECO will identify which packages are used by the EcoSpace and use reflection to obtain information about each of these packages; such as classes, relationships between classes, methods, and so on. An EcoSpace is capable of using only a single UML Package (as per these articles) or multiple UML Packages. This allows you to create generic UML Packages which may be reused across multiple projects; such as blogs, users, purchase ordering, and so on.

At runtime the EcoSpace provides multiple generic ECO services such as all persistence operations, in-memory transactions, OCL expression evaluation, and more. It also allows you to register your own service types which may then be used by your application, custom services may also be used by your business classes in an abstract way so that the classes are never tied to a single type of EcoSpace. In addition to these services the EcoSpace also contains all meta information about the model you created at design time, it is possible to discover information about classes, associations, association ends, and so on.

At design time reflection is used on the project's binaries in order to ascertain model information. This model information is used by the IDE integrated ECO designers in order to implement a richer development experience by providing services such as an OCL editor for defining OCL expressions.

Double-clicking the QuickStartEcoSpace.cs file will bring up a component editor for the EcoSpace. Along the bottom of the EcoSpace are a number of icons allowing you to perform a number of actions. These actions are:

Icon	Description
	Inspects the model for a number of common errors. Any errors will be displayed in the [Output] tab available from the View->Output menu item within Visual Studio.
	<p>Selects which UML Packages to use.</p> <p>ECO will inspect the project's References list for classes that are UML Packages and present them in a list allowing you to select or deselect them.</p> <p>Note that it is possible to have more than one type of EcoSpace in a single project, each using different UML Packages to form their model.</p>

Note: Other icons are included for backwards compatibility. The recommended approach is to use the PersistenceMapperProvider component instead.






On the design surface there is a single component "persistenceMapperSharer1". This component identifies which PersistenceMapperProvider to use.

PersistenceMapperProvider

The PersistenceMapperProvider is responsible for providing persistence operations to the EcoSpace. Only a single instance of the PersistenceMapperProvider is created at runtime, each instance of your EcoSpace uses the same PersistenceMapperProvider instance. The purpose of this component is to provide a single point for all persistence operations that is separate from the EcoSpace definition. It is possible to move the PersistenceMapperProvider out of this project completely and onto a separate machine, allowing multiple applications (WinForm, WebForm, etc) to access the same provider instance via remoting.

If you chose XML persistence in the ECO Wizard you will see a single component named "persistenceMapperXml1". This persistence component uses a local XML file to persist object instances. There are a number of alternative persistence options such as SqlServer, Oracle, MySql, Mimer, etc or you can implement your own.

At the bottom of the provider there are again a number of icons, these are:

Icon	Description
	Uses the persistence component to generate a structure suitable for holding instances of your business classes. In the case of a relational database it will create the necessary tables and columns.
	When the structure of your business classes changes this action will allow you to upgrade the structure of your data storage, renaming tables and columns, adding new tables, dropping tables for deleted classes and so on.
	Checks the model for errors, as per the EcoSpace.
	Normally ECO will create a number of support tables in the data storage identifying class->data mapping information. This action will create an XML file to hold the mapping information instead of adding the information to your data storage.
	This action is the opposite of the XML mapping action. It will ensure the mapping information is stored within the data storage.

Note: Allowing ECO to maintain your database structure is by far the most productive approach to writing ECO applications. When this is not an option it is also possible to define XML mapping information in order to have ECO use an existing database structure.

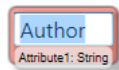
QuickStart.ASP / QuickStart.WinForm

These two projects are user-interface projects which consume the EcoSpace in order to manipulate instances of your modeled UML classes.

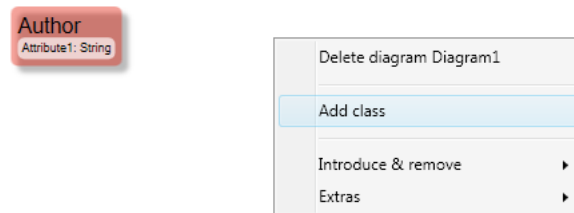
6 Creating a class diagram

Next a class will be added to the model.

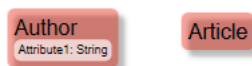
1. Double-click the file "QuickStart.ecomdl" to open the modeler.
2. A class named Class1 has been created automatically. We will now rename the class to "Author". This can be achieved by double-clicking the class on the drawing surface or in the Classes-treeview. This opens an editor where Class Name can be changed.



3. We will now create another class. Right-click the drawing surface and select Add Class. Name the new class "Article".



4. The final result should look something like this.



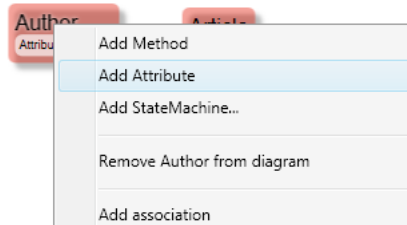
Exercise for the reader

Create a new class named "ArticleType". The class should have a "Name" attribute with a length of 32 characters; it should also have an "ID" attribute which is an AutoInc selected from the preset types menu.

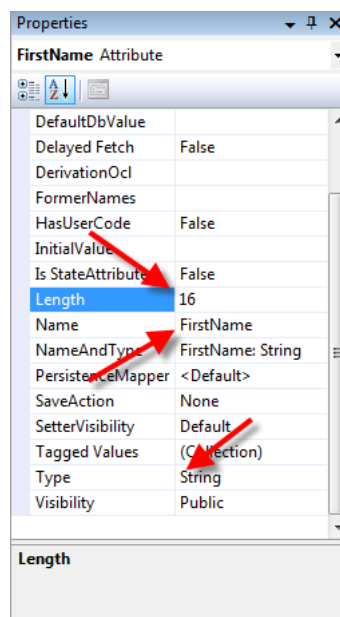
Note: You will need this class as these articles progress!

7 Adding attributes to the class

1. Right-click the Author class and select Add Attribute.



2. A window will appear when the property can be altered. Set the name to "FirstName" and select the type "String". Also change Length to "16".



3. The editor above is invoked by double-clicking a property in the class. Now add four additional attributes to the Author class. For the one of the attributes edit Attribute_1.

Name	Type	Length
EmailAddress	String	64
LastName	String	32
Password	String	32
Salutation	String	16
ID	AutoInc	

Note: The final attribute has the type "AutoInc" which is a special ECO type.

4. Now edit the class named "Article" and add the following UML attributes:

Name	Type	Length
Title	String	64
Description	String	255
Body	Text	-1
ID	AutoInc	

You may notice that the type of the "Body" attribute is "Text" rather than "String". This is a custom attribute type which will map in code to a string property on the class, if ECO is instructed to generate the database automatically it will create a column capable of storing a text/memo value. A length of -1 has been used to illustrate that there is no length limit, this is another custom ECO type. Custom types include:

Name	Delphi type	C# type	Description
AutoInc	Integer	int	Creates a table column that has a DB assigned auto-incrementing value.
Blob	TBytes	byte[]	Creates a table column that is capable of storing large binary data.
Image	TBytes	byte[]	Creates a table column that is capable of storing large binary data.
Text	string	string	Creates a table column that is capable of storing large textual data.

Your class diagram should now look something like this:



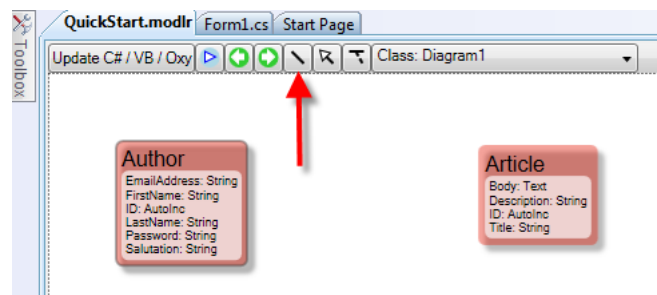
8 Adding associations between classes

An association is a way of allowing ECO classes to hold references to each other. Instead of adding a UML attribute to a class to hold a reference to another object a class association should be used because:

1. If the association is two-way (a property added to both classes on each end of the association) then the properties' object references at both ends will be kept in sync' with each other; setting `Article.Author` would also update `Author.Articles`.
2. It is possible to specify delete rules so that deleting an object on one end of the association has a specified effect on objects at the other end of the association; cascade delete, prohibit delete, or unlink.
3. Object instances at the other end of an association will be automatically fetched from the database when referenced via an association; `PurchaseOrder.Lines[0]` would automatically fetch the first `OrderLine` in the `PurchaseOrder`'s "Lines" association if it had not already been retrieved.
4. If ECO is instructed to create the database then many-to-many associations will result in a "link table" being created in the database.

Whenever one of your business classes needs to hold a reference to an instance of another ECO class you should always use an association. To add an association to the current model perform the following steps:

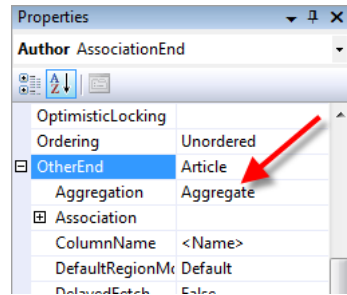
1. In the row of buttons on the class designer, select the "Association tool".



2. Now click on the Author class, hold the left mouse button down, and then move the mouse over to the Article class (as if you were dragging and dropping the Author class onto the Article class). As you move your mouse you will see an association line being drawn, to complete the operation release the left mouse button whilst the cursor is over the Article class, the association line will then connect to both classes.



3. Select the association by clicking on the line
4. In the Properties Window, locate the Aggregation property for the end called "Article" and set it to "Aggregated"



5. Your diagram should now look something like this. Note that it is possible to rearrange the role names and multiplicity labels independently.



Aggregation

UML aggregation in ECO it is a visual way of specifying the default delete behavior.

Aggregation	Appearance	Delete behavior
None		Objects at the aggregated end of the association (Article) are unlinked from the deleted object.
Aggregate		The object at the non-aggregated end of the association (Author) is prohibited from being deleted if any aggregated objects (Article) are associated with it.
Composite		Objects at the composite end of the association are considered to be "part of" the parent object, deleting the parent object (Author) will also delete its owned parts (Article).

Note that this is the default behavior, it is still possible to specify a different delete action for association ends "Delete action" drop down list in the [End 1] and [End 2] tabs.

Since this model will be in english, we can specify a default suffix to add to association ends that contain more than one object. Switch to the Modlr-tab and focus on the package node ("QuickStart"). In the properties window, set the "PluralSuffix" to "s". This will automatically set the name of the end currently called "Article" to "Articles" since it can contain more than a single article. We can always override the plural-behaviour for an association end if we have relations to a class called for example "Party" ("Parties").

Exercise for the reader

Now add an association between the Article and ArticleType classes. Use the correct aggregation kind so that an ArticleType may not be deleted if any articles are associated with it. The association should permit an ArticleType to have

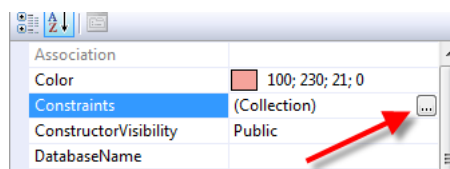
many Article instances and an Article to have only a single ArticleType. Use the correct aggregation kind to prevent the ArticleType from being deleted if there are any Articles associated with it.

9 Adding object validation

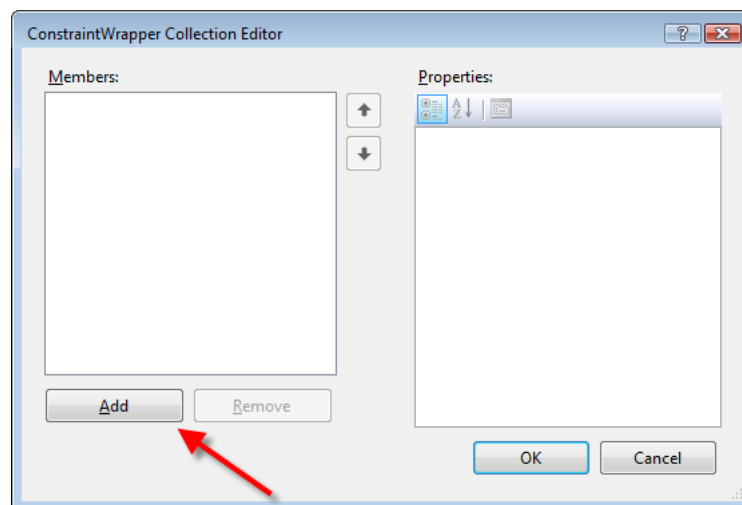
ECO allows you to specify a list of OCL expressions against the classes in your model, during runtime it is possible to obtain a list of modeled constraints and evaluate their expressions in order to determine whether or not the current object instance passes validation. The main benefit of having validation constraints in the model is that it becomes very easy to ensure consistent validation is available in your application no matter which part of your application performs the validation; the same validation rules will apply to your business model whether you have developed a WinForm, ASP .NET, or web service interface.

To add constraints to your model follow these steps:

1. Bring up the class editor by selecting Article class.
2. In the Properties Window, open the collection editor for the property Constraints

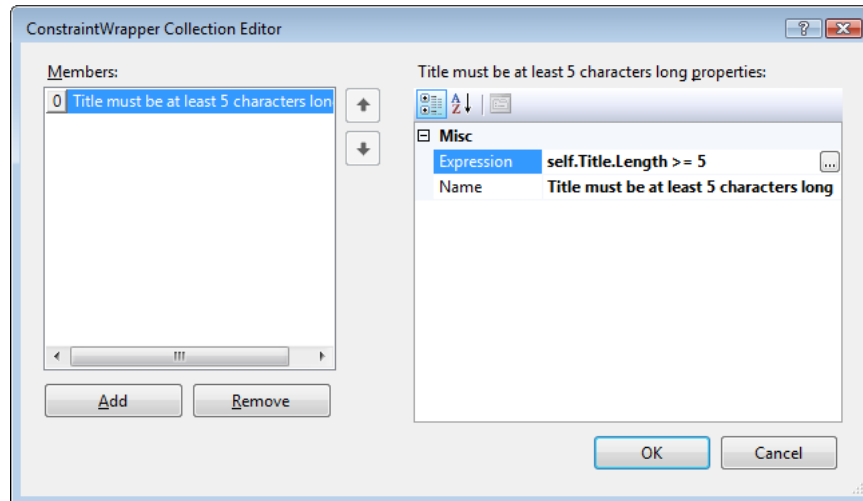


3. Click the "Add" button below the (empty) list of constraints.



4. Edit the properties of the new constraint:

```
Name = Title must be at least 5 characters long  
Expression = self.Title.Length >= 5
```



Note: There is an OCL editor dialog available by clicking the elipsis button next to the expression property to help you to create valid OCL expressions.

Now add the following additional constraints to the Article class, click the "Insert" button at the top left of the form to add a new row to the grid for each constraint:

Name	OCL Expression
Description must be at least 10 characters long	self.Description.Length >= 10
Body must be at least 50 characters long	self.Body.Length >= 50
Author required	self.Author->NotEmpty

Note: "self" is case-sensitive. "self" is not mandatory unless the member name clashes with a class name, in this case "self.Author" is required to ensure the OCL evaluator understands that you want to identify the "Author" property of the current object and not the class "Author".

Now add the following constraints to the Author class, you may want to copy / paste the OCL expression for the last entry to ensure it is correct:

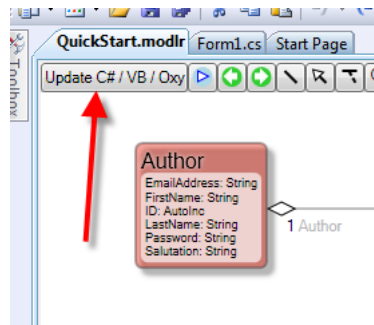
Name	OCL Expression
Salutation must be at least 2 characters	self.Salutation.Length >= 2
First name must be at least 2 characters	self.FirstName.Length >= 2
Last name must be at least 2 characters	self.LastName.Length >= 2
Password must be at least 6 characters	self.Password.Length >= 6
Email address required	self.EmailAddress.Length > 0
Invalid email address	(self.EmailAddress.length = 0) or (self.EmailAddress.regExpMatch("\\w+([-.]\\w+)*@\\w+([-.]\\w+)*\\.\\w+([-.]\\w+)*"))

The Author constraints will reveal that the "Email address required" is invalid if no email address is entered, and that the "Invalid email address" is invalid if an email address is entered but it is not a valid email address. The regular expression was obtained from <http://www.regexpalib.com> - the expression was then "escaped" to make it valid OCL by preceding each

backslash with an additional backslash, this is because backslash in OCL is a special character just as it is in C# for example.

10 Generating source code

A simple business model has now been created, however, it currently only exists as a design. In order to implement the design source code must first be generated. This is done by selecting the "Tools" menu, and then "Generate Eco Model".



Now that the source has been generated, rebuild the solution to produce its DLL file. ECO designers use reflection to read model information during application development, so it is advisable to ensure the project containing the model is compiled each time you have finish making modifications and have updated your model's source code.

10.1 Inspecting the source code

Open the source code that has been generated. The first class to inspect is the class for the package itself (QuickStart.cs/QuickStart.eco.cs). The class is identified as a Package using a reflection attribute named "UmlElement".

The class in QuickStart.cs appears to be completely empty:

```
// This file contains any user-written code for the package

namespace QuickStart {
    using [...]

    public abstract partial class QuickStartPackage {
    }
}
```

but in QuickStart.eco.cs we find some code that looks more interesting:

```
namespace QuickStart {
    using [...]

    [UmlElement("Package", Id="a4bec50f-f294-4b68-9fc6-e2dc9cfb40d5")]
    [UmlMetaAttribute("ownedElement", typeof(QuickStart.ArticleType))]
    [UmlMetaAttribute("ownedElement", typeof(QuickStart.Article))]
    [UmlMetaAttribute("ownedElement", typeof(QuickStart.Author))]
    public abstract partial class QuickStartPackage {

        [UmlElement("Association", Id="8fc408cf-36ee-4431-a4db-6dd95c7aa60c")]
        public class ArticleArticleArticleTypeArticleType {
        }

        [UmlElement("Association", Id="f9870d94-aa67-43a3-828e-24116d016c4e")]
        public class ArticleArticleAuthorAuthor {
        }
    }
}
```

```

    }
}

```

Each of the classes belonging to this package are listed above the class and identified using a reflection attribute named "UmlMetaAttribute", this enables ECO to determine which business classes belong to the package at runtime by using reflection. Within the class itself there are nested classes within a region named "Associations", there will be one nested per association within your model. In this case the association from Author to Article was automatically named ArticleArticleAuthorAuthor. An embedded class is created for each association in order to provide a place for ECO to generate reflection attributes holding information such as whether the association is persistent or transient (does not get saved to the persistent storage).

Now inspect the "Author.cs" source code. Again we see a completely empty class, but when we inspect Author.eco.cs we find the framework generated code. Above the Author class itself you will see a collection of reflection attributes describing meta information from the model.

```

[UmlElement(Id="d72188e8-2113-4107-aa7f-af36894ff340")]
[UmlMetaAttribute("constraint", "Salutation must be at least 2
characters=self.Salutation.Length >= 2")]
[UmlMetaAttribute("constraint", "First name must be at least 2
characters=self.FirstName.Length >= 2")]
[UmlMetaAttribute("constraint", "Last name must be at least 2
characters=self.LastName.Length >= 2")]
[UmlMetaAttribute("constraint", "Password must be at least 6
characters=self.Password.Length >= 6")]
[UmlMetaAttribute("constraint", "Email address required=self.EmailAddress.Length > 0")]
[UmlMetaAttribute("constraint",
    "Invalid email address=(self.EmailAddress.length = 0)
or\n(self.EmailAddress.regExpMatch('\\\\\\w+([-+.]\\\\\\\\w+)*@\\\\\\\\w+([-+.]\\\\\\\\w+)*\\\\\\\\.\\\\\\\\w+([-+.]\\\\\\\\w+)*\\\\\\\\')"))]

```

The class is uniquely identified using the reflection attribute "UmlElement" with a GUID. In addition you will see the constraints that were added earlier in this document. These reflection attributes are also used to hold other information such as ECO state machines, so this information will grow as your model approaches completion.

All of the code in the file Author.eco.cs will be generated whenever you make changes to your class in the model. Never make any manual changes to this code. User written code should always be placed in Author.cs.

```

public partial class Author : Eco.ObjectImplementation.ILoopBack2,
    System.ComponentModel.INotifyPropertyChanged
{
    [...]
}

```

We see that our class implements the two interfaces ILoopback2 and INotifyPropertyChanged. The first interface is a framework specific interface that lets the framework read and write data in the class. The second interface is a .net specified interface that lets UI controls subscribe to changes in the object (used in WPF applications)

The first part of the class defines an embedded class named "Eco_LoopbackIndices". This class contains a set of constants that identify the members of the Author class by index. This index is used internally by ECO. In addition it provides a reliable way of accessing model information at runtime (not covered in this article).

```

public class Eco_LoopbackIndices {
    public const int Eco_FirstMember = 0;
    public const int Salutation = Eco_FirstMember;
    public const int ID = (Salutation + 1);
    public const int FirstName = (ID + 1);
    public const int LastName = (FirstName + 1);
}

```

```

public const int Password = (LastName + 1);
public const int EmailAddress = (Password + 1);
public const int Article = (EmailAddress + 1);
public const int Eco_MemberCount = (Article + 1);
}

```

The following section contains some ECO support source code:

```

public virtual void set_MemberByIndex(int index, object value)
public virtual object get_MemberByIndex(int index)
Eco.ObjectRepresentation.IObject Eco.ObjectRepresentation.IObjectProvider.AsIObject()
void Eco.ObjectImplementation.ILoopBack2.SetContent(
    Eco.ObjectImplementation.IContent content)

```

- **AsIObject:** Provides access to ECO framework functions for any object. For example `SomePerson.AsIObject().Delete` would mark the object for deletion. The `IObject` interface is used throughout the ECO framework. `IObject` is ECO's way of referring to object instances in an abstract way.
- **set_MemberByIndex / get_MemberByIndex:** Once an instance of a modeled class is created or retrieved from the data storage OCL expressions will read its property values directly from a local cache (the "EcoSpace"). When the developer needs to implement code within the property they will mark the property `HasUserCode=True` in the model. Any property that is marked in such a way will automatically have additional source code generated within these methods to enable the OCL evaluator to read/write values using the property accessors directly rather than having to restore to reflection.

Each UML attribute will generate a property on the class, along with a read/write method. The "get" method will return the cached value from the `EcoSpace`, the "set" method will set the value within the `EcoSpace` cache. If you are familiar with patterns you might recognize this as the "facade" pattern.

Finally a property is declared with the information from the model, information such as the maximum allowed length of the property.

```

[UmlElement(Id="4e95409a-5fa7-4d26-98f6-9674dcc6b171",
Index=Eco_LoopbackIndices.Salutation)]
[UmlTaggedValue("Eco.Length", "16")]
public string Salutation {
    get {
        return
        ((string)(this.eco_Content.get_MemberByIndex(Eco_LoopbackIndices.Salutation)));
    }
    set {
        this.eco_Content.set_MemberByIndex(Eco_LoopbackIndices.Salutation,
        ((object)(value)));
    }
}

```

Each association will also generate a property on the class, returning either a single instance of another class if the association end was modeled as singular, or a collection of instances if the association was modeled as multiple. In this example source code for the `Author.Articles` role (association end) has been generated. You will see additional model information such as

1. The name of the association. This is the association class nested within the package class.
2. The multiplicity of the role.
3. The aggregation type used.

```
[UmlElement("AssociationEnd", Id="fa4dd884-1402-49d7-8c25-71e4f6f9bf02",
Index=Eco_LoopbackIndices.Article)]
[UmlMetaAttribute("aggregation", AggregationKind.Aggregate)]
[UmlMetaAttribute("association",
typeof(QuickStartPackage.ArticleArticleAuthorAuthor), Index=1)]
[UmlMetaAttribute("multiplicity", "1..*")]
public IEcoList<Article> Article {
    get {
        return
((IEcoList<Article>)(this.eco_Content.get_MemberByIndex(Eco_LoopbackIndices.Article)));
    }
}
```

Again this list may grow as your model becomes more complete.

Finally, you may notice the following two constructors in the class (taken from Article.cs):

```
public Author(Eco.ObjectRepresentation.IEcoServiceProvider serviceProvider)
protected Author(Eco.ObjectImplementation.IContent content)
```

The first constructor is executed when an entirely new instance is created.

The second constructor is executed when the object is recreated from the data storage. This occurs when a previously created object is saved, the application terminates, and then the object instance is retrieved when the application next runs.

11 Derived members

Derived members allow the developer to create attributes or associations in the business model that are calculated from other members in the business model. These members may be calculated in the following ways:

Type	Description
OCL derived	An OCL expression is entered in the business model for the derived member, when the value of this member is requested the OCL expression is evaluated and a result is produced.
Code derived	The developer creates a method on the class with a specific name and parameters, when the value of this member is requested the method is executed and returns the calculated value.
Reverse derived	For reading a value this is exactly the same as a normal code derived member. In addition the programmer is able to implement an additional method which is responsible for storing the reverse-engineered values elsewhere. For example, setting a reverse derived member "Age: Integer" would adjust the year in "DateOfBirth : DateTime".

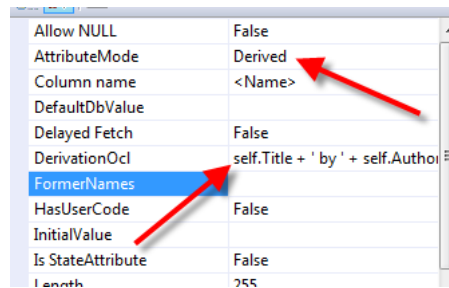
When the value of a derived member is first requested ECO will either evaluate the specified OCL expression or execute the method written by the developer and then return the result. This result will then be automatically cached within the EcoSpace and any subsequent requests will return the same value from the cache, saving the application from having to recalculate the value each time; when the calculations involved are intensive this can result in a vastly improved user experience.

To ensure the cached value does not become stale ECO employs a subscription approach (observer pattern). A change notification request is placed on each element that is used to derive the result. When an element value changes the cached value is discarded, requesting the derived value after its cached value has been discarded will result in the value being recalculated and cached again. Note that derived values are only ever calculated on demand and if there is no previously calculated cache value stored, this improves performance again by ensuring calculations only occur when needed and as infrequently as possible.

11.1 Creating OCL derived members

1. Open the design surface for the class diagram.
2. Right-click the Article class and add a new UML attribute.
3. Name the attribute "Summary" and specify its type as "String".
4. Set its AttributeMode to "Derived".
5. Set its "DerivationOcl" property to

```
self.Title + ' by ' + self.Author.AsString
```

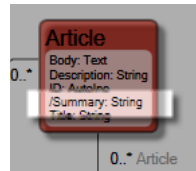


7. Now regenerate source code for the model.

8. Compile the project.

9. Save all changes.

When creating an OCL derived association ECO will automatically place the required subscriptions on each of the elements used to create the resulting value.



11.2 Creating code derived members

If a derived attribute doesn't specify any OCL, it is assumed to have its derivation logic implemented in code.

1. Open the design surface for the class diagram.
2. Right-click the Author class and add a new UML attribute (Add property).
3. Name it "FullName" and specify its type as "String".
4. Set its AttributeMode property to "Derived", but this time do not specify a "Derivation OCL" value.
5. Generate source code for your model.
6. Open the file Author.cs (Remember, don't use Author.eco.cs).
7. Insert the following function in the (currently empty) class Author:

```
private string FullNameDerive()
{
    return Salutation + " " + FirstName + " " + LastName;
}
```

The method should return an object of the correct type for the property on the class; a string in this case. This method looks as though it will evaluate the result every time `SomePersonInstance.FullName` is read but this is not the case. To illustrate the caching and subscription behavior in ECO we will have a quick delve into the WinForm application.

1. Open the source code for Form1.cs in the QuickStart.WinForms project.
2. In the constructor add the following code:

```
InitializeComponent();

rhRoot.EcoSpace = ecoSpace;
QuickStart.Author a = new QuickStart.Author(EcoSpace);
a.Salutation = "Mr";
a.FirstName = "Peter";
a.LastName = "Morris";
MessageBox.Show(a.FullName + " you just experienced a break");
MessageBox.Show(a.FullName + " no breakpoint this time");
a.FirstName = "Oliver";
MessageBox.Show(a.FullName + " the value changed, so you experienced a breakpoint");
```

3. Add a breakpoint to the FullNameDerive method.
 4. Run the WinForm application.
- First the breakpoint will pause the application execution showing that FullNameDerive() is being executed, followed by a message box telling you that you just experienced a breakpoint.
 - Next another message box will appear informing you that you did not experience a breakpoint. This shows that the value returned last time was cached and there was no need to execute this method.
 - Next the breakpoint will pause the application again to evaluate the result of FullNameDerive(), this is because the FirstName was changed to "Oliver".
 - Finally a message box will show you the new value of FullName "Mr Oliver Morris".

How does this work?

This is one of the many advanced techniques that may be achieved by having classes read and write their state from a local cache. Internally ECO uses its `IAutoSubscriptionService` to track all "elements" that are accessed when your method is executed. Elements are properties of objects (UML attributes or association ends), the existence state of object instances (whether or not they are deleted), and also class extents (a notification when a new object instance is created). The auto subscription is initialized before executing your method, and finally stops auto-subscribing once it is exited. This means that all accessed elements are subscribed to no matter how they are accessed, either by calling any number of other methods, or even calling other derived members.

When a derived member needs to read the value of another derived member, for example if `Article.Summary` was derived as

```
self.Title + " by " + self.Author.FullName
```

the auto subscription service handles the subscriptions separately, for example when reading `SomeArticle.Summary`.

1. Create a new `AutoSubscriptionContext`.
 1. Read `self.Title` + subscribe.
 2. Read `self.Author` + subscribe.
 3. read `self.Author.FullName` + subscribe.
 1. Create a new `AutoSubscriptionContext`

1. Read self.Salutation + subscribe.
2. Read self.FirstName + subscribe.
3. Read self.LastName + subscribe.

2. Remove the AutoSubscriptionContext.

2. Remove the AutoSubscriptionContext.

Using the `IAutoSubscriptionService` it is possible to obtain a reference to the current `AutoSubscriptionContext` so that you may manually add additional subscriptions to non-ECO events such as a file on the hard disk changing. The following code is a bit "deep" for an article at such an early stage in this series so don't worry if you don't understand the example, it is here to illustrate a feature rather than to show how to implement something.

```
IAutoSubscriptionService autoSubscribe =
    this.AsIOObject().ServiceProvider.GetEcoService<IAutoSubscriptionService>();

MyFileWatcher fileWatcher = new MyFileWatcher(this.FolderPath);
fileWatcher.AddSubscriber(autoSubscribe.ActiveSubscriber);
```

"MyFileWatcher" is a fictitious class I have used to illustrate an instance of a class that is nothing to do with ECO and therefore cannot be automatically subscribed to (the values of its properties are not stored in the `EcoSpace` cache). "FolderPath" is a string UML attribute on your class and identifies which folder to monitor. Here is an incomplete example of how such a class would handle the ECO part of its source code.

```
private List<ISubscriber> Subscribers = new List<ISubscriber>();

public void AddSubscriber(ISubscriber subscriber)
{
    Subscribers.Add(subscriber);
}

private void FileNotification()
{
    for (int index = Subscribers.Count - 1; index >= 0; index--)
    {
        ISubscriber currentSubscriber = Subscribers[index];
        if (currentSubscriber.IsAlive)
            currentSubscriber.Receive(this, EventArgs.Empty);
        else
            Subscribers.RemoveAt(index);
    }
}
```

The rest of the class would require you to use a `FileSystemWatcher` (part of the .NET framework) to watch the folder specified and execute `FileNotification` when appropriate. It isn't recommended that you attempt such an implementation at this stage. Understanding the ability was the goal, if you understand the source code that is a bonus.

11.3 Creating derived settable members

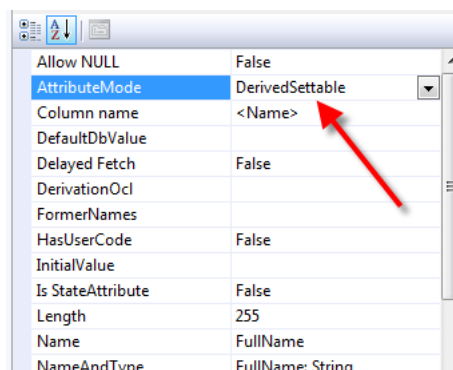
Unlike standard UML attributes / association ends marked with `HasUserCode=True` a derived member will by default be read-only, the generated source code will only have getter code. A derived settable member is the same as a code or OCL derived member except it also allows the user to modify the calculated value. The generated code will now additionally have some property setter code. As the value cannot be stored in the derived member itself it is the developer's responsibility to parse the entered value and update the relevant members elsewhere.

Here are some more scenarios in which you may wish to use this feature:

1. Allowing a user to enter their current age, and then adjusting their DateOfBirth to indicate in which year they were born.
2. Allowing a user to adjust the gross amount on an invoice resulting in the net and tax amounts being adjusted automatically.
3. Allowing DepartureTime, ArrivalTime, and TripDuration to be modified, where TripDuration is derived, and for all three to be kept in sync.

To allow the user to modify a derived attribute follow these steps:

1. Open the editor for the property "FullName".
2. Set the AttributeMode to "DerivedSettable".



4. Generate source code for your model.
5. Open the source code for the Author class in the editor (Author.cs, not Author.eco.cs).
6. Add the following method to the class Author:

```
partial void FullNameReverseDeriving(string value)
{
    string[] nameParts = value.Split(new char[] { '/' });
    if (nameParts.Length == 3)
    {
        if (nameParts[0].Length != 0)
        {
            Salutation = nameParts[0];
        }
        if (nameParts[1].Length != 0)
        {
            FirstName = nameParts[1];
        }
        if (nameParts[2].Length != 0)
        {
            LastName = nameParts[2];
        }
    }
}
```

The user may now enter an author's name by modifying the FullName in the format "Salutation/FirstName/LastName" instead of having to set focus to three separate controls and enter each part individually. If there are not exactly three parts to the user entry (two forward slashes) then the input is ignored. If any of the values entered are empty then the current

value will be left. For example, the value "//Morris" would change only the last name.

12 Default string representation

In the previous example "Creating OCL derived members (see page 23)", in the OCL expression you will see the text "self.Author.AsString". Each modeled business class has a "Default String Representation" property which is an OCL expression that should be evaluated for when the OCL identifier "AsString" is evaluated. This would mean that the OCL previously used for Article Summary

```
self.Title + ' by ' + self.Author.AsString
```

would result in a string looking something like this

```
MyFirstArticle by CapableObjects.QuickStart.BusinessModel.Author
```

This is because by default the Default String Representation will return the name of the class. Instead the model will be changed to show the author's full name using our code-derived "FullName" derived attribute.

1. Open the design surface for the class diagram and open the edit window for the Author class
2. In the ECO tab, set the "Default String Representation" to "self.FullName".
3. Edit the Article class and set the "Default String Representation" to "self.Title + ' by ' + self.Author.AsString".
4. Generate source code for your model.
5. Compile the project and save all changes.

13 Summary

This article has described how to use the integrated modeler to create classes, UML attributes, and associations. This model may now not only be used in various applications but may also be combined with other packages or even extended upon by other packages. These are all topics that will be covered later in this series.

Index

A

- Adding associations between classes 13
- Adding attributes to the class 11
- Adding object validation 16

C

- Creating a class diagram 10
- Creating an ECO model 5
- Creating code derived members 24
- Creating derived settable members 26
- Creating OCL derived members 23

D

- Default string representation 29
- Derived members 23

G

- Generating source code 19

I

- Inspecting the source code 19

O

- Overview 1

P

- Prerequisites and goals 2

S

- Summary 30

T

- Terminology in ECO and UML 3

U

- Understanding the project template 7